# Project Plan (1st Draft)

**Team Dec 15-12: @PaniniJ**
**February 19th, 2015**

|  |  |  |
|---|---|---|
| **Advisor** | Dr. Rajan | |
| **Client** | Dr. Rajan | |
| **Team Members** | Dalton Mills | *Webmaster* |
| | David Johnston | *Team Lead* |
| | Kristin Clemens | *Communication Lead* |
| | Trey Erenberger | *Key Concept Holder* |

# Table of Contents

## Project Statement

Modern multi-core CPUs provide powerful support for concurrent programs, but it is very hard to build correct programs and systems which take full advantage of this hardware. PaniniJ is a Java-like programming language with strong support for the capsule programming abstraction. The language has been designed such that programs written in the language cannot include certain common concurrency bugs.

Though `panc`, the existing PaniniJ compiler, is an extension of the standard Java compiler, PaniniJ code is not easily integrated into existing Java projects. Our project, @PaniniJ, intends to make Capsule-Oriented Programming more accessible to Java programmers than the existing PaniniJ tools can provide.

Our project will implement an alternative mechanism for generating capsules, one which can be used by a Java developer within the Java programming language and built using standard Java build systems (e.g. Ant, Maven, Eclipse and `javac`). Our solution will use annotation processors to easily generate capsule classes which are functionally equivalent to PaniniJ capsules but usable from within Java.

## Prior Work: Background on Capsule-Oriented Programming

TODO: Existing concurrent programming techniques require a developer to ...
TODO: Actor frame of reference?
TODO: Our system will existing theoretical work which has been done on Capsule

## Usage: Capsule Generation via Annotation Processor

Annotation processors provide a simple way to hook into a standards-compliant java compiler, inspect the client’s annotated Java code, and then automatically construct additional Java source files. These machine-generated source files are then compiled and included for use in other source code in the project.

Our solution will be distributed as a `.jar` file, which a client should easily be able to include in their java project. The user can then write a Java class annotated with `@Capsule`. This original `@Capsule`-annotated class is referred to as a *template class* because it serves as the template from which a capsule class is generated. When this class is compiled, `javac` will call on our annotation processor. Our annotation processor will inspect the template class, and generate a capsule class which wraps the template class appropriately.

## Requirements

### Functional Requirements

- Working annotation-based frontend for the generation of PaniniJ-compatible capsules.
- Support for four basic execution strategies (i.e. Thread, Task, Monitor, and Sequential).
- Reference-counting garbage collection.
- Translate PaniniJ programs to @PaniniJ programs for use as
    - capsule-generation test cases,
    - semantic analysis test cases, and
    - benchmarks comparing PaniniJ and @PaniniJ performance.
- Intra-capsule confinement analysis.
- Analysis of other yet-unidentified safety/correctness properties.
- Automated mapping from code to a compilation strategy

### Nonfunctional Requirements

- Project Requirements
- Fits Java programmer's expectations
- each requirement should have a Validation and Acceptance Test

## Possible Solutions

There are several options that we came across that would allow us to make capsule programming more accessible

- Create an Eclipse plugin for PaniniJ
    - Effectively build an IDE on top of Eclipse in order to support the PaniniJ language.
    - Made easier through the use of Xtext, a library for creating Eclipse plugins and language development easier.
- Take Eclipse JDT's existing ASTParser to allow it to build an AST for PaniniJ
    - This option was very briefly researched for reasons below.
- Leverage new annotation processing and pluggable type checkers in Java 1.8 to plug directly into the standard java compiler.

We opted for the third option of creating a library which uses Java 1.8's annotation processing. The other options were fairly quickly discarded in favor of this approach for the reasons outlined below:

1. Annotation Processing is now standard in Java 1.8. We would not be creating a non-standard solution.
2. Creating a standard solution caters to our ultimate goal of having accessible capsule-oriented programming.
3. It will be easier to integrate into existing projects.

## Risk & Feasibility Assessment

Risk reduction was the ultimate deciding factor in why we chose to use annotation processing for this project. Moving forward, however, we still have many concerns regarding potential and inherent risks of our proposed solution, as well as questions about the feasibility of achieving our goal of accessible capsule-oriented programming.

### Performance

1. How will multiple stages of code generation and type checking affect compiler performance?
2. Will code generated by the annotation processor be as performant as `panc` generated code?

### Workflow Integration

1. Will @PaniniJ provide sufficiently familiar development workflow for the average developer programming concurrent solutions?
2. @PaniniJ template and capsule classes are just Java code, but what will be required from the user to make common Java IDEs work seamlessly with @PaniniJ?
3. Will IDE debugger integration be possible? If so, will it work consistently across all IDEs for which @PaniniJ is implemented?

### Attainment of Project Goals

1. Will @PaniniJ lower the barrier to entry for developers new to programming concurrent solutions?

### Semantic Analysis

1. Will @PaniniJ be able to provide the semantic analysis expected by experienced IDE users? If not, how much will this negatively impact the user's ability to program effectively using @PaniniJ, and thus their desire to use @PaniniJ at all?

## Schedule

By the end of the Spring 2015 semester:

- Working annotation-based frontend for the generation of PaniniJ-compatible capsules.
- Support for four basic execution strategies (i.e. Thread, Task, Monitor, and Sequential).
- Translate a number of existing PaniniJ benchmark programs into @PaniniJ for use as tests.
- Reference-counting garbage collection.

By the end of the Fall 2015 semester:

- Translate more existing PaniniJ benchmark programs into @PaniniJ for use as tests and benchmarks comparing the performance of PaniniJ and @PaniniJ systems.
- Implement intra-capsule confinement analysis.
- Implement semantic analysis of additional safety/correctness properties.
- Automated selection of appropriate execution policy based on system topology.

## Cost

This project does not have any hardware components and does not require the purchase of any commercial software. By building on top of research undergone at Iowa State University we receive a vast pool of resources for no cost. We therefore expect our project budget to be $0 and since we are working for free, the cost of development will be $0.

## Market Survey/Literature Review

Our final product will compete with existing PaniniJ tools created within the Laboratory for Software Design at Iowa State University, in particular, the `panc` compiler. This existing solution has several restrictions that negatively affect its usability for the average Java programmer. Our project seeks to resolve some of these shortcomings.

PaniniJ currently inhibits the programmer from using standard Java toolchains. A client needs to use the special `panc` compiler. However, our solution is easily integrated via the classpath by hooking into the standard Java compiler produced by Oracle. `panc` forks and extends the Java compiler produced by Oracle. Our product decouples the capsule-generation process from a specific version of compiler allowing for portability between toolchains. Our products are comparable because they utilize the same for capsule oriented programming and produce outputs compatible with the PaniniJ runtime.

## Conclusion

Our project's main goal is providing java developers with a simple way to write capsule-based concurrent programs using the Panini capsule oriented paradigm. This is achieved by produced a deliverable Jar that easily integrates with standard Java workflow and toolchain.

If our project is successful and we implement enough functionality to allow our product to compete with the existing implementation, it is likely that our product will replace the existing `panc` compiler and the Panini paradigm will use annotated Java over its own language syntax. This ultimately results in the Laboratory of Software Design migrating to our project and building upon it to implement further features of Panini for Java. After our work is completed, the project will be released as an open source project on Sourceforge or Github and will be maintained and contributed to by the Panini research group.

## Glossary

| | |
|---|---|
| Panini | A capsule-oriented programming language whose goals are to ease development of correct, scalable, and portable concurrent software.[1] |
| PaniniJ | A capsule-oriented extension of the Java programming language that runs on the standard JVM platform.[2] |
| Capsule | Similar to a process, it defines a set of public operations and also serves as a memory region for some set of ordinary objects.[3] |
| Capsule-oriented Programming | A programming paradigm that aims to ease development of concurrent software systems by allowing abstraction from concurrency-related concerns; entails breaking down program logic into distinct parts called *capsule* declarations and composing these parts to form the complete program using *system* declaration.[4] |
| Java annotations | A form of metadata, provide data about a program that is not part of the program itself, having no direct effect on the operation of the code they annotate.[5] |
| Java type annotations | Annotations that are applied to *type use*, which support improved analysis of Java programs way of ensuring stronger type checking.[6] |
| Pluggable type checker | A custom module used in conjunction with the Java compiler to ensure the state of a variable, with the goal of preventing or detecting all errors of a given variety. |
| Checker Framework | A pluggable type checker tool developed at The University of Washington[7] that comes with predefined type checkers and provides a framework for developers to build their own custom type checkers. |

[1] http://www.cs.iastate.edu/~panini/about.shtml#what
[2] http://www.cs.iastate.edu/~panini/about.shtml#what
[3] http://www.cs.iastate.edu/~panini/about.shtml#capsule
[4] http://www.cs.iastate.edu/~panini/docs/faq.shtml#q_l_whatis
[5] http://docs.oracle.com/javase/tutorial/java/annotations/index.html
[6] http://docs.oracle.com/javase/tutorial/java/annotations/type_annotations.html
[7] http://types.cs.washington.edu/checker-framework/